**Динамические структуры данных** – это структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый *динамическим распределением памяти*, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных характеризуется тем что:

* она не имеет имени;
* ей выделяется память в процессе выполнения программы;
* количество элементов структуры может не фиксироваться;
* размерность структуры может меняться в процессе выполнения программы;
* в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Каждой динамической структуре данных сопоставляется статическая переменная типа указатель (ее значение – адрес этого объекта), посредством которой осуществляется доступ к динамической структуре.

Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели – это статические величины, поэтому они требуют описания.

Необходимость в динамических структурах данных обычно возникает в следующих случаях.

* Используются переменные, имеющие довольно большой размер (например, массивы большой размерности), необходимые в одних частях программы и совершенно не нужные в других.
* В процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем.
* Когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Динамические структуры, по определению, характеризуются отсутствием физической смежности элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется *связным*.

Достоинства связного представления данных – в возможности обеспечения значительной изменчивости структур:

* размер структуры ограничивается только доступным объемом машинной памяти;
* при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
* большая гибкость структуры.

Вместе с тем, связное представление не лишено и недостатков, основными из которых являются следующие:

* на поля, содержащие указатели для связывания элементов друг с другом, расходуется дополнительная память;
* доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива – с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Порядок работы с динамическими структурами данных следующий:

1. создать (отвести место в динамической памяти);
2. работать при помощи указателя;
3. удалить (освободить занятое структурой место).

**Классификация динамических структур данных**

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

* однонаправленные (односвязные) списки;
* двунаправленные (двусвязные) списки;
* циклические списки;
* стек;
* дек;
* очередь;
* бинарные деревья.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки.

Каждая компонента любой динамической структуры представляет собой запись, содержащую, по крайней мере, два поля: одно поле типа указатель, а второе – для размещения данных. В общем случае запись может содержать не один, а несколько указателей и несколько полей данных. Поле данных может быть переменной, массивом или структурой. Для наилучшего представления изобразим отдельную компоненту в виде:

http://www.intuit.ru/EDI/18_07_16_2/1468794052-15773/tutorial/909/objects/28/files/28_001.png

где поле Р – указатель; поле D – данные.

Элемент динамической структуры состоит из двух полей:

* *информационного поля* (поля данных), в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой – вектором, массивом, другой динамической структурой и т.п.;
* *адресного поля* (поля связок), в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;

Объявление элемента динамической структуры данных выглядит следующим образом:

struct имя\_типа {

информационное поле;

адресное поле;

};

Например:

struct TNode {

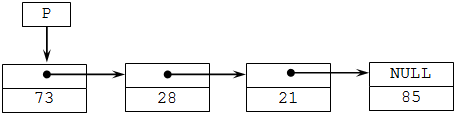
int Data;//информационное поле

TNode \*Next;//адресное поле

};

Информационных и адресных полей может быть как одно, так и несколько.

Рассмотрим в качестве примера динамическую структуру, схематично указанную на [рис. 28.1](http://www.intuit.ru/studies/courses/648/504/lecture/11455?page=2#image.28.1):



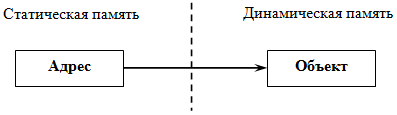
**Рис. 28.1.** Схематичное представление динамической структуры

Данная структура состоит из 4 элементов. Ее первый элемент имеет поле Data, равное 73, и связан с помощью своего поля Next со вторым элементом, поле Data которого равно 28, и так далее до последнего, четвертого элемента, поле Data которого равно 85, а поле Next равно NULL, то есть нулевому адресу, что является признаком завершения структуры. Здесь P является указателем, который указывает на первый элемент структуры.

**Доступ к данным в динамических структурах**

Элемент динамической структуры в каждый момент может либо существовать, либо отсутствовать в памяти, поэтому его называют *динамическим*. Поскольку элементами динамической структуры являются динамические переменные, то единственным средством доступа к динамическим структурам и их элементам является указатель (адрес) на место их текущего расположения в памяти. Таким образом, доступ к динамическим данным выполняется специальным образом с помощью указателей.

Указатель содержит адрес определенного объекта в динамической памяти. Адрес формируется из двух слов: адрес сегмента и смещение. Сам указатель является статическим объектом и расположен в сегменте данных ( [рис. 28.2](http://www.intuit.ru/studies/courses/648/504/lecture/11455?page=2#image.28.2)).



**Рис. 28.2.** Связь указателя с адресуемым объектом

Для обращения к динамической структуре достаточно хранить в памяти адрес первого элемента структуры. Поскольку каждый элемент динамической структуры хранит адрес следующего за ним элемента, можно, двигаясь от начального элемента по адресам, получить доступ к любому элементу данной структуры.

Доступ к данным в динамических структурах осуществляется с помощью операции "стрелка" ( -> ), которую называют операцией косвенного выбора элемента структурного объекта, адресуемого указателем. Она обеспечивает доступ к элементу структуры через адресующий ее указатель того же структурного типа. Формат применения данной операции следующий:

УказательНаСтруктуру-> ИмяЭлемента

Операции "стрелка" ( -> ) двуместная. Применяется для доступа к элементу, задаваемому правым операндом, той структуры, которую адресует левый операнд. В качестве левого операнда должен быть указатель на структуру, а в качестве правого – имя элемента этой структуры.

Например:

p->Data;

p->Next;

Имея возможность явного манипулирования с указателями, которые могут располагаться как вне структуры, так и "внутри" отдельных ее элементов, можно создавать в памяти различные структуры.

Однако необходимо помнить, что работа с динамическими данными замедляет выполнение программы, поскольку доступ к величине происходит в два шага: сначала ищется указатель, затем по нему – величина.

**Работа с памятью при использовании динамических структур**

В программах, в которых необходимо использовать динамические структуры данных, работа с памятью происходит стандартным образом. Выделение динамической памяти производится с помощью операции new или с помощью библиотечной функции malloc (calloc). Освобождение динамической памяти осуществляется операцией delete или функцией free.

Например, объявим динамическую структуру данных с именем Node с полями Name, Value и Next, выделим память под указатель на структуру, присвоим значения элементам структуры и освободим память.

struct Node {char \*Name;

int Value;

Node \*Next

};

Node \*PNode; //объявляется указатель

PNode = new Node; //выделяется память

PNode->Name = "STO"; //присваиваются значения

PNode->Value = 28;

PNode->Next = NULL;

delete PNode; // освобождение памяти

**Ключевые термины**

**Адрес сегмента** – это одно из машинных слов, составляющих адрес динамического элемента, которое представляет собой адрес первого элемента структуры.

**Адресное поле (поле связок)** – это поле структуры, в котором содержится указатель, связывающий данный элемент с другими элементами структуры.

**Динамические структуры данных** – это структуры данных, память под которые выделяется и освобождается не на этапе компиляции, а в процессе работы программы.

**Динамический элемент** – это элемент динамической структуры, который в конкретный момент выполнения программы может либо существовать, либо отсутствовать в памяти.

**Динамическое распределение памяти** – это выделение памяти под отдельные элементы в тот момент, когда они "начинают существовать" в процессе выполнения программы.

**Информационное поле (поле данных)** – это поле структуры, в котором содержатся непосредственно обрабатываемые данные.

**Связное представление данных** – это установление связи между элементами динамической структуры через указатели.

**Смещение** – это одно из машинных слов, составляющих адрес динамического элемента, которое представляет собой изменение адреса относительно первого элемента структуры.

**Динамические структуры данных.**

**Стеки, очереди. Списки. Бинарные деревья.**

[Очереди 1](#_Toc217872016)

[Стеки 3](#_Toc217872017)

[Списки 6](#_Toc217872018)

[Бинарные деревья 7](#_Toc217872019)

**Динамические структуры данных** – это структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый динамическим распределением памяти, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Порядок работы с динамическими структурами данных следующий:

* создать (отвести место в динамической памяти);
* работать при помощи указателя;
* удалить (освободить занятое структурой место).

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

* однонаправленные (односвязные) списки;
* двунаправленные (двусвязные) списки;
* циклические списки;
* стек;
* дек;
* очередь;
* бинарные деревья.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки.

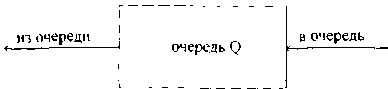
# Очереди

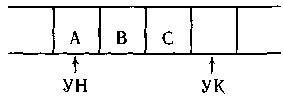
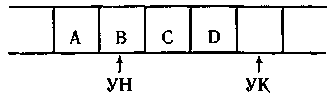
**Очередью** называют структуру данных, для которой определены операции добавления и удаления элементов и элементы которой организованы таким образом, что их удаление будет осуществляться точно в таком же порядке, в каком происходило их добавление.

Очередь представляет собой линейный список данных, доступ к которому осуществляется по принципу "первый вошел, первый вышел" (иногда сокращенно его называют методом доступа FIFO).

В повседневной жизни очереди встречаются часто. Например, очередь в банке или очередь в кафетериях с быстрым обслуживанием являются очередью в указанном выше смысле (исключая те случае, когда человек пытается добиться обслуживания вне очереди!).

Программным образом такая структура может быть реализована различными способами. Простейшей является реализация с помощью массива и двух указателей. Один из указателей определяет то место массива, куда элемент будет добавляться, и называется ***концом*** очереди (УК). Другой указатель определяет то место массива, откуда элемент будет удаляться, и называется ***началом*** очереди (УН). Условием наличия элементов в очереди является неравенство указателей УН и УК. При этом порядок элементов в очереди определен таким образом, что элемент с меньшим индексом поступил раньше.



Например, пусть у нас есть очередь из трех элементов А,В,С, в которую первым был помещен элемент А, затем — элемент В, а последним — элемент С. После удаления элемента А первым элементом очереди будет элемент В. При этом элемент А остался в массиве на своем месте, но его уже нет в очереди, так как указатель начала очереди УН указывает на элемент В, т. е. следующий элемент массива. Если мы добавим в очередь новый элемент D, то он будет помещен в конец очереди, определяемый указателем конца УК. При этом значение указателя УК должно измениться. Следует отметить, что для того, чтобы удалить из очереди элемент С, необходимо сначала удалить элемент В. Очередь, в которой нет ни одного элемента, называется *пустой.* В этом случае оба указателя показывают на одно и то же место. **Операции над очередями: Init** — создает пустую очередь; **Empty** — возвращает значение true, если очередь пуста, и false в противном случае; **Insert** — добавляет элемент в конец очереди; **Remove** — удаляет элемент из начала очереди.

**Const** maxqueue=5;

**Type** Queue=array[1..maxqueue] **of** integer;

**var**

q : Queue;

First,Free,code,op : integer;

i,n,x,k : integer;

**Procedure Init** (var q: Queue;

**var** Free, First : integer);

**begin**

First:=1;

Free:=1;

**end**;

**Function Empty** (var q: Queue; **var** First, Free:integer): boolean;

**begin**

**if** First=Free **then** Empty:=true

**else** Empty:=false;

**end**;

**Procedure InsQue** (**var** q: Queue; **var** Free,First: integer; **var** Code: integer;

**var** Op: integer; x: integer);

**begin**

**if** (Op=1) **and** (Free=First) **then**

**begin**

Code:=1; {Очередь полна}

exit;

**end**;

Op:=1;

q[Free]:=x;

Free:=Free+1;

**If** Free>maxqueue **then** Free:=1;

Code:=0;

**end**;

**Procedure RemQue**(**var** q: Queue; **var** Free,First:integer; **var** Code: integer;

**var** Op: integer; **var** x: integer);

**begin**

**if** (Op=0) **and** (Free=First) **then**

**begin**

Code:=2; {Очередь пуста}

exit;

**end**;

Op:=0;

x:=q[First];

First:=First+1;

**if** First>maxqueue **then** First:=1;

Code:=0;

**end**;

**Begin**

**Init**(q,first,Free);

**Empty**(q,first,free);

**if** Empty(q,first,free)=true **then**

**while** code=0 **do**

**begin**

read(x);

**InsQue**(q,free,first,code,op,x);

writeln;

**for** i:=1 **to** maxqueue **do**

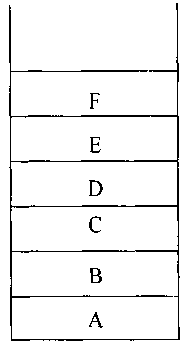
write (q[i],' ');

**end**;

**end.**

Очереди в программировании используются во многих случаях, например, при моделировании, при планировании работ (метод ПЕРТ или графики Гатта), при буферизации ввода-вывода.

# Стеки

**Стеком** будем называть структуру данных, для которой определены операции добавления и удаления элементов и элементы которой организованы таким образом, что их удаление будет осуществляться точно противоположно тому порядку, в каком проводилось добавление.

Организация стека в определенном смысле противоположна организации очереди, поскольку здесь используется доступ по принципу "последней пошел, первый вышел" (такой метод доступа иногда называют методом LIFO). Представим себе стопку тарелок. Нижняя тарелка из этой стопки будет использована последней, а верхняя тарелка (которая была установлена в стопку последней) будет использована первой. Стеки широко используются в системном программном обеспечении, включая компиляторы и интерпретаторы.

Представленный здесь стек содержит 6 элементов, причем элементы в стек были помещены в следующем порядке: А, В, С, D, E, F. Извлекаться же они могут только в порядке F, E, D, С, В, А, т. е. для того, чтобы извлечь из стека элемент С, необходимо сначала извлечь элементы F, Е и D.

**Операции над стеками и их реализация: Init** — создает пустой стек; **Empty** — возвращает значение true, если стек пуст и false в противном случае; **Push** — добавляет элемент в стек; **Pop** — удаляет элемент из стека. Исторически сложилось так, что две основные операции для стека - поместить в стек и выбрать из стека - получили название соответственно "затолкнуть" и "вытолкнуть".

**Const** Maxstack=3;

**Type**

Stack = **Array**[1..Maxstack] **of** integer;

**var**

i,tos,x,code : integer;

s:Stack;

**Procedure Init**(**var** s: Stack; **var** tos : integer);

**begin**

tos:=0;

**end**;

**Function Empty** (tos : integer): **boolean**;

**begin**

**if** tos = 0 **then** Empty:=true

**else** Empty:=false;

**end**;

**Procedure Push**(**var** s: Stack; **var** tos: integer; **var** Code: integer; x: integer);

**begin**

**if** tos = Maxstack **then**

**begin**

Code:= 1;

exit;

**end**;

tos:=tos + 1;

s[tos]:=x;

Code:=0;

**end**;

**Procedure Pop**(**var** s: Stack; **var** tos: integer; **var** Code: integer; **var** x: integer);

**begin**

**if** Empty (tos) **then**

**begin**

Code:=2;

exit;

**end**;

x:=s[tos];

tos:=tos-1;

Code:=0;

**end**;

**Begin**

**Init**(s,tos);

**Empty**(tos);

**if** Empty(tos)=true **then**

**while** code=0 **do**

**begin**

read(x);

Push(s,tos,code,x);

writeln;

**for** i:=1 **to** Maxstack **do** write (s[i],' ');

**end**;

**end.**

Хорошим примером применения стека является калькулятор, который может выполнять четыре действия. Большинство калькуляторов используют стандартную форму выражений, которая носит название инфиксной формы. В общем виде ее можно представить в виде "операнд-оператор-операнд". Например, для прибавления 100 к 200 вы должны ввести число 100, набрать символ сложения, ввести число 200 и нажать клавишу со знаком равенства. Однако, некоторые калькуляторы применяют другую форму выражений, получившую название постфиксной формы. В этом случае оба операнда вводятся перед вводом оператора. Например, для добавления 100 к 200 при использовании постфиксной формы сначала вводится число 100, затем вводится число 200 и после этого нажимается клавиша со знаком плюс. Введенные операнды помещаются в стек. При вводе оператора из стека выбираются два операнда и результат помещается в стек. При использовании постфиксной формы очень сложные выражения могут легко вычисляться на калькуляторе.

Ниже показана программа для такого калькулятора.

{ калькулятор с четырьмя операциями, иллюстрирующий работу }

**program** four\_function\_calc;

**const**

MAX = 100;

**var**

stack:array [1..100] of integer;

tos:integer; { указатель вершины стека }

a, b:integer;

s:string[80];

{ поместить объект в стек }

**procedure** Push(i:integer);

**begin**

**if** tos >= MAX **then** Writeln('Stack full') **else**

**begin**

stack[tos] :=i;

tos := tos+1;

**end**;

**end**;{Push}

{ выборка объекта из стека }

**function** Pop:integer;

**begin**

tos := tos-1;

**if** tos < 1 **then**

**begin**

Writeln('Stack underflow')

tos := tos+1;

Pop := 0;

**end** **else** Pop := stack[tos];

**end**;{ Pop }

**begin** { калькулятор }

tos := 1;

Writeln('For Function Calculator');

**repeat**

Write(': '); { вывод приглашения }

Readln(s);

Val(s, a, b) { преобразование строки символов в целое число }

{ считается, что при успешном преобразовании пользователь ввел число,

а в противном случае пользователь ввел оператор }

**if** (b=0) **and** ((Length(s)>1) **or** (s[1]<>'-')) **then**

Push(a) **else**

**case** s[1] **of**

'+' : **begin**

a := Pop;

b := Pop;

Writeln(a+b);

Push(a+b);

**end**;

'-' : **begin**

a := Pop;

b := Pop;

Writeln(b-a);

Push(b-a);

**end**;

'\*' : **begin**

a := Pop;

b := Pop;

Writeln(a\*b);

Push(a\*b);

**end**;

'/' : **begin**

a := Pop;

b := Pop;

**if** a=0 **then** Writeln('divide by zero')

**else**

**begin**

Writeln(b div a);

Push(b div a);

**end**;

**end**;

'.' : **begin**

a := Pop;

Writeln(a);

Push(a);

**end**;

**end**;

**until** UpCase(s[1])='G'

**end**.

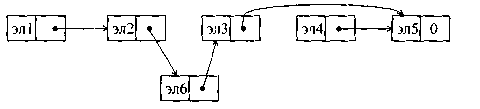
Для того, чтобы посмотреть, что находится в вершине стека, достаточно ввести точку. Хотя данная программа выполняет арифметические действия только с целыми числами, ее легко можно приспособить для чисел с плавающей запятой, изменяя тип данных стека и преобразуя оператор "div" в оператор деления для чисел с плавающей запятой.

2+2\*2+21/7\*2+100/5-12+(-10)\*2 = 0

2 2 2\*+2 21 7/\*+ 100 5/+12 -10 2\*+ = 0

# Списки

**Списком** будем называть некоторую последовательность элементов, связанных посредством ***указателей*** (ссылок).

В списке элементы связаны друг с другом логически. Логический порядок следования элементов списка определяется с помощью указателей. Логический порядок следования элементов списка может не совпадать с физическим порядком их расположения в памяти ЭВМ. Каждый элемент списка (часто называемый узлом) состоит из двух частей. Первая часть содержит непосредственно данные, принадлежащие элементу, и является информационной. Вторая часть содержит указатель и является справочной. Указатель определяет место положения элемента списка, связанного с данным элементом.

Связанные списки используются в двух основных случаях. Во-первых, при создании массивов, которые располагаются в оперативной памяти и размер которых заранее неизвестен. Если вы заранее знаете, какого размера память потребуется для решения вашей задачи, то вы можете использовать простой массив. Однако, если действительный размер списка вам неизвестен, то вы должны применить связанный список. Во-вторых, связанные списки используются в базах данных на дисках. Связанный список позволяет быстро выполнять вставку и удаление элемента данных без реорганизации всего дискового файла. По этим причинам связанные списки широко используются в программах по управлению базами данных.

Связанные списки могут иметь одиночные или двойные связи. Список с одной связью содержит элементы, каждый из которых имеет связь со следующим элементом данных. В списке с двойной связью каждый элемент имеет связь как со следующим элементом, так и с предыдущим элементом. Тип связанного списка выбирается в зависимости от решаемой задачи.

**Операции над списками:** Для ***перехода*** к очередному элементу используется указатель на следующий элемент списка. ***Поиск*** осуществляется путем последовательного просмотра элементов списка до тех пор, пока искомый элемент не найден или пока не окончен список. Если мы дошли до конца списка, то искомого элемента в списке нет. ***Операция вставки*** нового элемента и ***операция удаления*** элемента. В отличие от очередей и стеков элемент может быть добавлен в список в любое место и удален с любого места с минимальными затратами. На рисунке показана вставка элемента эл6 и удаление элемента эл4.

***Формирование пустого списка.*** Заводим в памяти двумерный массив. В Sp[l,i] будем хранить сам элемент списка, а в Sp[2,i] — указатель на следующий элемент списка.

const maxspisok = 20;

Type Spisok=array of [1..2,l..maxspisok] of integer;

Затем определяются значения указателей для пустого списка. Переменная US будет задавать адрес первого элемента списка свободных мест, а переменная UN — адрес первого элемента списка. Для пустого списка US = 1, a UN = 0.

**Procedure** NewSpisok (**var** Sp:Spisok; **var** US,UN:integer;);

**var** i: integer;

**begin**

**for** i:=l **to** Maxspisok-1 **do** Sp[2,i]: = i+1;

Sp[2,Maxspisok]: = 0;

US:=1;

UN: = 0;

**end**;

***Добавление элемента Х в список.*** В промежуточной переменной US2 запоминаем адрес первого элемента списка свободных мест. Помещаем по этому адресу значение X, и в качестве указателя на следующий элемент списка помещаем номер первого элемента списка. Теперь указателем первого элемента списка будет указатель US, так как по этому адресу поместили новый элемент. Он из разряда свободных элементов переходит в разряд занятых. Указателем списка свободных мест будет US2, т. е. элемент, который следует в списке свободных мест за только что исключенным из него элементом.

**Procedure** InsSpisok(**var** Sp: Spisok; **var** US,UN:integer;

**var** Code: integer; **var** x:integer);

**var** US2: integer;

**begin**

**if** US = 0 **then**

**begin**

Code:= 1;

exit;

**end**;

US2: = Sp[2,US];

Sp[l,US]: = X;

Sp[2,US]: = UN;

UN: = US;

US: = US2;

Code: = 0;

**end**;

***Поиск элемента Х в списке.*** Переменной IND присваиваем значение указателя первого элемента списка. Просматриваем список до тех пор, пока не будет найден элемент X или пока не достигнем последнего элемента списка (IND = 0). Если IND = 0, то в списке нет элемента со значением X.

**Procedure** PoiskX(**var** Sp: Spisok; **var** US,UN:integer;

**var** x:integer; **var** Code: integer; **var** Pred,Ind: integer);

**begin**

Ind: = UN;

Pred: = 0;

**while** (Sp[l,Ind]<>X) **and** (Ind<>0) **do**

**begin**

Pred: = Ind;

Ind: = Sp[2,Ind];

**end**;

**if** Ind = 0 **then** Code: = 2 **else** Code:=0;

**end**;

***Удаление элемента X из списка.*** Сначала находим элемент X в списке (выполняется процедура PoiskX) и только после этого, если Code = 0, мы можем его удалить. При удалении изменяется указатель записи элемента, идущего в списке перед элементом X, таким образом, чтобы он показывал на элемент, идущий в списке после элемента X. Так как позиция, в которой располагается удаляемый элемент, переходит из разряда занятых в разряд свободных, то мы помещаем эту позицию в список свободных мест. Теперь это будет первая позиция списка свободных мест.

**Procedure** RemSpisokX(**var** Sp: Spisok; **var** Code : integer;

**var** US,UN:integer; Pred,Ind:integer);

**begin**

Sp[2,Pred]: = Sp[2,Ind];

Sp[2,Ind]: = Us;

US: = Ind;

Code: = 0;

**end**;

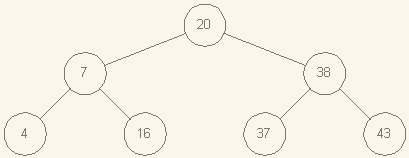
# Бинарные деревья

**Бинарное (двоичное) дерево (binary tree)** - это упорядоченное дерево, каждая вершина которого имеет не более двух поддеревьев, причем для каждого узла выполняется правило: в левом поддереве содержатся только ключи, имеющие значения, меньшие, чем значение данного узла, а в правом поддереве содержатся только ключи, имеющие значения, большие, чем значение данного узла.

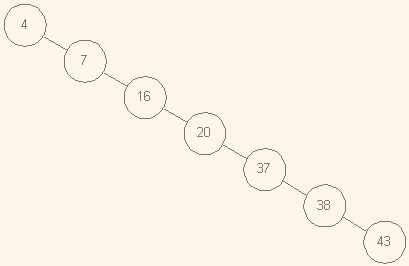
Бинарное дерево является рекурсивной структурой, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева.

Узел дерева, не имеющий потомков, называется листом.

Схематичное изображение бинарного дерева:



Бинарное дерево может представлять собой пустое множество, и может выродиться в список. Вырожденное бинарное дерево:



**Операции над бинарными деревьями**

Бинарное дерево должно реализовывать следующие операции:

1. Инициализация бинарного дерева: текущий указатель устанавливается неопределенным (или нулевым, nil), а количество узлов нулевым.
2. Помещение в бинарное дерево элемента: для нового элемента в бинарном дереве создается соответствующий узел, указатели на потомков которого пусты (поиск позиции для такого узла начинается с корня и проходит согласно правилам, определяющим структуру бинарного дерева).
3. Получение значения текущего элемента
4. Поиск заданного элемента: если искомый элемент находится в дереве, то возвращается указатель на него, в противном случае возвращается nil, сигнализирующий о неуспехе поиска значения
5. Удаление узла из дерева
6. Уничтожение бинарного дерева

**Рассмотрим эти операции более подробно:**

***Структура для создания корня и узлов дерева имеет вид:***

**type**

T = Integer;{ скрываем зависимость от конкретного типа данных }

TTree = ^TNode;

TNode = **record**

value: T;

Left, Right: TTree;

**end**;

Здесь поля Left и Right - это указатели на потомков данного узла, а поле value предназначено для хранения информации.

При ***создании дерева*** вызывается рекурсивная процедура следующего вида:

**procedure** Insert(var Root: TTree; X: T);

{ Дополнительная процедура, создающая и инициализирующая новый узел }

**procedure** CreateNode(var p: TTree; n: T);

**begin**

New(p);

p^.value := n;

p^.Left := nil;

p^.Right := nil

**end**;

**begin**

**if** Root = **nil** **Then** CreateNode(Root, X) { создаем новый узел дерева }

**else**

**with** Root^ **do**

**begin**

**if** value < X **then** Insert(Right, X) **else**

**if** value > X **Then** Insert(Left, X) **else**

{ Действия, производимые в случае повторного

внесения элементов в дерево}

**end**;

**end**;

Эта процедура добавляет элемент X к дереву, учитывая величину X. При этом создается новый узел дерева.

***Получить значение текущего элемента*** можно так:

**function** GetNode(Root: TTree): T;

**begin**

**if** Root = nil **then** WriteLn('Дерево - пусто!')

**else**

GetNode:=Root^.value

**end**;

***Поиск заданного элемента*** (функция возвращает адрес узла с найденным элементом; если элемент в дереве не найден, возвращается nil):

**function** Find(Root: TTree; X: T): TTree;

**begin**

**if** Root = nil **then** Find := nil **else**

**if** X = Root^.value **then** Find := Root **else**

**if** X < Root^.value **then** Find := Find(Root^.Left, X) **else**

Find := Find(Root^.Right, X);

**end**;

***Удаление узла бинарного дерева.***

Это немного более сложная операция, чем поиск заданного элемента. Сначала необходимо найти элемент, подлежащий удалению. Если этот элемент - лист, он может быть просто удален. Если же он является внутренним узлом (имеет потомков), то просто так удалить его не получится - будут разрушены внутренние связи в дереве.

Поступаем так:

если удаляемый узел имеет только одного "сына", то его значение можно заменить значением этого "сына"

если у удаляемого элемента 2 "сына", заменяем его элементом с наименьшим значением среди потомков правого "сына" (или элементом с наибольшим значением среди потомков левого "сына")

Для реализации процедуры Remove желательно иметь функцию DeleteMin, которая будет удалять наименьший элемент непустого дерева Root, и возвращать значение удаленного элемента:

**function** DeleteMin(var Root: TTree): T;

**var** WasRoot: TTree;

**begin**

**if** Root^.Left = nil **then**

**begin**

DeleteMin := Root^.value;{ Результат функции }

WasRoot := Root;{ Запоминаем узел для последующего удаления }

Root := Root^.Right;{ продвигаемся дальше }

Dispose(WasRoot);{ удаляем бывший корень }

**end else** { узел Root имеет левого "сына" }

DeleteMin := DeleteMin(Root^.Left);

**end**;

Теперь процедура Remove может быть реализована так:

**procedure** Remove(var Root: TTree; X: T);

**var** WasNext: TTree;

**begin**

**if** Root <> **nil** **then**

**if** X < Root^.value **then** Remove(Root^.Left, X) **else**

**if** X > Root^.value **then** Remove(Root^.Right, X) **else**

**if** (Root^.Left = nil) **and** (Root^.Right = nil) **then**

**begin**

{ Нет "сыновей", удаляем узел, на который указывает Root }

Dispose(Root);

Root := nil

**end else**

**if** Root^.Left = **nil** **then**

**begin**

WasNext := Root^.Right;

Dispose(Root);

Root := WasNext;

**end else**

**if** Root^.Right = **nil** **then**

**begin**

WasNext := Root^.Left;

Dispose(Root);

Root := WasNext;

**end else** { у удаляемого элемента есть оба "сына" }

Root^.value := DeleteMin(Root^.Right);

**end**;

***Уничтожение бинарного дерева.***

**Procedure** Delete(T: TTree);

**Begin**

**If** T = nil **Then** Exit;

Delete(T^.Right);

Delete(T^.Left);

Dispose(T)

**End**;

***Обход дерева***

Прохождение (или обход) бинарного дерева означает систематическое перечисление всех узлов для выполнения необходимой функциональной обработки. Наиболее известны и практически важны 3 способа прохождения, которые отличаются порядком и направлением обхода бинарного дерева. Можно проходить узлы в прямом порядке (сверху-вниз), в симметричном порядке (слева-направо) и, наконец, в концевом порядке (снизу-вверх).

Рекурсивные алгоритмы прохождения бинарного дерева по каждому из перечисленных способов включают 3 одинаковых процедуры, где нужно пройти корень поддерева, левое поддерево текущего корня и правое поддерево текущего корня. Направление обхода однозначно определяет последовательность выполнения указанных процедур. Последовательность их рекурсивного вызова для каждого способа прохождения перечислена в следующей таблице.

Таблица рекурсивных алгоритмов прохождения бинарного дерева

|  |  |  |
| --- | --- | --- |
| **Порядок прохождения** | | |
| **Прямой** | **Симметричный** | **Концевой** |
| 1. Корень поддерева  2. Левое поддерево  3. Правое поддерево | 1. Левое поддерево  2. Корень поддерева  3. Правое поддерево | 1. Левое поддерево  2. Правое поддерево  3. Корень поддерева |

Прямой порядок прохождения означает обход в направлении сверху-вниз, когда после посещения очередного разветвления продолжается прохождение вглубь дерева, пока не пройдены все потомки достигнутого узла. По этой причине прямой порядок прохождения часто называют нисходящим, или прохождением в глубину.

Прямой порядок есть наиболее естественный способ перечисления узлов дерева в форме вложенных скобок или уступчатого списка. Если исключить все скобки и запятые, то последовательность узлов в форме вложенных скобок будет соответствовать прямому порядку прохождения дерева.

Уступы в ступенчатом списке, представляющем любую иерархическую структуру, также располагаются в прямом порядке. В генеалогических терминах прямой порядок прохождения дерева отражает династический порядок престолонаследования, когда титул передается старшему потомку.

При симметричном порядке дерево проходится слева-направо, порождая лексиграфически упорядоченную последовательность ключей узлов. По этой причине симметричный порядок прохождения часто называют лексиграфическим. Симметричность порядка выражается в том, что если бинарное дерево отразить относительно вертикальной оси, поменяв местами левые и правые узлы, то симметричный порядок прохождения заменится на противоположный лексиграфический.

Если применяется концевой порядок прохождения, то получается обход дерева снизу-вверх, когда в момент посещения любого узла все его потомки уже пройдены, а корень дерева проходится последним. Из-за этой особенности обхода, концевой порядок часто называют восходящим, или обратным относительно прямого.

При выборе определенного порядка прохождения требуемая функциональная обработка каждого узла происходит после соответствующего числа заходов в него. При обходе сверху-вниз каждый узел обрабатывается при первичном заходе в него, при симметричном порядке прохождения - во втором, а при обходе снизу-вверх - в третьем.

Еще один вариант обхода - по уровням: сначала выводить корень, потом - все узлы первого уровня, за ними - узлы второго уровня, и т.д.

Это реализуется вот такой процедурой:

**procedure** PrintByLevel(level: integer; **var** items: array of TTree; count: integer);

**var** i, new\_count: integer;

**begin**

**if** count <> 0 **then**

**begin**

writeln('level = ', level);

new\_count := 0;

**for** i := 0 **to** pred(count) **do**

**begin**

write(items[i]^.value:4);

**if** items[i]^.left <> **nil** **then**

**begin**

inc(new\_count);

items[count + new\_count - 1] := items[i]^.left;

**end**;

**if** items[i]^.right <> **nil** **then**

**begin**

inc(new\_count);

items[count + new\_count - 1] := items[i]^.right;

**end**;

**end**;

writeln;

move(items[count], items[0], new\_count\*sizeof(TTree));

PrintByLevel(level + 1, items, new\_count);

**end**;

**end**;

Вызывать процедуру надо вот так:

**var**

arr: array[0 .. pred(size)] of TTree; { <--- Здесь должно быть достаточно места для хранения }

**begin**

{ Заполнение дерева }

...

arr[0] := root;

PrintByLevel(0, arr, 1);

...

**end**.

**Графическое представление бинарного дерева**

Для отрисовки бинарного дерева в графическом режиме можно использовать процедуру PrintTreeGraph.Примечание: приведенная процедура работает с деревьями, состоящими из символов (T = Char) или строк (T = String). Для того, чтобы она работала с другими типами, необходимо изменить функцию

**Function** Convert(X: T): String;

так, чтобы она преобразовывала необходимый тип к строке...

**uses** graph;

**type**

T = Char;

PTree = ^TTree;

TTree = **record**

value: T;

Left, Right: PTree;

**end**;

**procedure** PrintTreeGraph(Root: PTree); { Заменить при необходимости }

**function** Convert(X: T): **string**;

**begin**

Convert := X

**end**;

**var** start\_x, start\_y: integer;

**const**

delx = 30;

dely = 20;

circle\_r = 10;

btw = circle\_r div 2;

**procedure** print\_node(Root: PTree; Level: Integer; L, C, R: Integer);

**function** min(a, b: Integer): Integer;

**begin**

min := a;

**if** b < a **then** min := b

**end**;

**function** Center(a, b: Integer): Integer;

**begin**

Center := min( a, B ) + abs( a - B ) div 2;

**end**;

**var** pos\_y: integer;

**begin**

pos\_y := start\_y + pred(level) \* dely;

**if** Root^.Right <> **nil** **then**

**begin**

Line(C, pos\_y, Center(C, R), pos\_y + dely);

{ вот тут мы предотвращаем наложение (уходим чуть вправо) }

print\_node(root^.right, Level + 1, C+btw, Center(C+btw, R-btw), R-btw);

**end**;

**if** Root^.Left <> **nil** **then**

**begin**

Line(C, pos\_y, Center(L, C), pos\_y + dely);

{ и вот тут тоже (уходим левее) }

print\_node(Root^.Left, Level + 1, L+btw, Center(L+btw, C-btw), C-btw);

**end**;

SetColor(Black);

SetFillStyle(SolidFill, Black);

PieSlice(C, pos\_y, 0, 359, circle\_r);

SetColor(White);

Circle(C, pos\_y, circle\_r);

SetTextJustify(CenterText, CenterText);

OutTextXY(C, pos\_y, Convert(Root^.value));

**end**;

**begin**

start\_x := GetMaxX div 2;

start\_y := 10;

print\_node(Root, 1, 0, GetMaxX div 2, GetMaxX);

**end**;

**Применение бинарных деревьев**

Организация данных с помощью бинарных деревьев часто позволяет значительно сократить время поиска нужного элемента. Поиск элемента в линейных структурах данных обычно осуществляется путем последовательного перебора всех элементов, присутствующих в данной структуре. Поиск по дереву не требует перебора всех элементов, поэтому занимает значительно меньше времени. Максимальное число шагов при поиске по дереву равно высоте данного дерева, т.е. количеству уровней в иерархической структуре дерева.

Следующий модуль содержит основные функции для работы с бинарными деревьями.

**unit** TreeUnit;

**interface**

**type**

T = Integer;

TTree = ^TNode;

TNode = **record**

value: T;

Left, Right: TTree;

**end**;

**procedure** Insert(var Root: TTree; X: T);

**procedure** Remove(var Root: TTree; X: T);

**function** Find(Root: TTree; X: T): TTree;

**procedure** Delete(Root: TTree);

**procedure** LeafsCount(Root: TTree; **var** k: Integer);

**function** GetNode(Root: TTree): T;

**procedure** PrintDown(Root: TTree);

**procedure** PrintLex(Root: TTree);

**procedure** PrintUp(Root: TTree);

**implementation**

**procedure** Insert(var Root: TTree; X: T);

**procedure** CreateNode(var p: TTree; n: T);

**begin**

New(p);

p^.value := n;

p^.Left := nil;

p^.Right := **nil**

**end**;

**begin**

**if** Root = **nil** **then**

CreateNode(Root, X) **else**

**with** Root^ **do**

**begin**

**if** value < X **then** Insert(Right, X) **else**

**if** value > X **then** Insert(Left, X)

**end**;

**end**;

**function** DeleteMin(var Root: TTree): T;

**var** WasRoot: TTree;

**begin**

**if** Root^.Left = **nil** **then**

**begin**

DeleteMin := Root^.value;

WasRoot := Root; { запоминаем узел для последующего удаления }

Root := Root^.Right; { продвигаемся дальше }

Dispose(WasRoot); { удаляем бывший корень }

**end** **else** DeleteMin := DeleteMin(Root^.Left);

**end**;

**procedure** Remove(**var** Root: TTree; X: T);

**var** WasNext: TTree;

**begin**

**if** Root <> **nil** **then**

**if** X < Root^.value **then** remove(Root^.Left, X) **else**

**if** X > Root^.value **then** remove(Root^.Right, X) **else**

**if** (Root^.Left = **nil**) **and** (Root^.Right = **nil**) **then**

**begin**

Dispose(Root); { Освобождаем выдыленную Root-у память }

Root := nil { ОбNILение узла, на всякий случай }

**end** **else**

**if** Root^.Left = **nil** **then**

**begin**

WasNext := Root^.Right;

Dispose(Root);

Root := WasNext;

**end** **else**

**if** Root^.Right = **nil** **then**

**begin**

WasNext := Root^.Left;

Dispose(Root);

Root := WasNext;

**end** **else** Root^.value := DeleteMin(Root^.Right);

**end**;

**procedure** Delete(Root: TTree);

**begin**

**if** Root = **nil** **then** exit;

Delete(Root^.Right);

Delete(Root^.Left);

Dispose(Root)

**end**;

**function** Find(Root: TTree; X: T): TTree;

**var** p: TTree;

**begin**

**if** Root <> **nil** **then**

**begin**

p := Root;

**while** p <> **nil** **do**

**begin**

**if** X < p^.value **then** p := p^.Left **else**

**if** X = p^.value **then** Break **else** p := p^.Right

**end**;

Find := p

**end** **else** Find := nil

**end**;

**procedure** PrintDown(Root: TTree);

**begin**

**if** Root = **nil** **then** exit;

**with** Root^ **do**

**begin**

Writeln(value, '':2);

PrintDown(Left);

PrintDown(Right)

**end**

**end**;

**procedure** PrintLex(Root: TTree);

**begin**

**if** Root = **nil** **then** exit;

**with** Root^ **do**

**begin**

PrintLex(Left);

Writeln(value, '':2);

PrintLex(Right)

**end**

**end**;

**procedure** PrintUp(Root: TTree);

**begin**

**if** Root = **nil** **then** exit;

**with** Root^ **do**

**begin**

PrintUp(Left);

PrintUp(Right);

Writeln(value, '':2);

**end**

**end**;

**procedure** LeafsCount(Root: TTree; **var** k: Integer);

**begin**

**if** Root <> **nil** **then**

**begin**

LeafsCount(Root^.Left, k);

**if** (Root^.Left = nil) **and** (Root^.Right = nil) **then** Inc(k);

LeafsCount(Root^.Right, k)

**end**

**end**;

**function** GetNode(Root: TTree): T;

**begin**

**if** Root = **nil** **then**

WriteLn('Error - tree is empty !')

**else** GetNode:=Root^.value

**end**;

**end**.

Кроме основных функций в модуле содержится процедура подсчета числа "листьев" - узлов, не содержащих потомков (количество листьев возвращается в переменной k):

Procedure LeafsCount(T: TTree; Var k: Integer);

Процедуры обхода:

procedure PrintDown(T: TTree);{ Прямой порядок прохождения }

procedure PrintLex(T: TTree);{ Симметричный порядок прохождения }

procedure PrintUp(T: TTree);{ Концевой порядок прохождения }

Пример программы, использующей реализацию бинарного дерева:

**uses** TreeUnit;

**const**

size = 10;

iV: **array**[1 .. size] **of** Integer =

( 1, 4, 8, 2, 7, 4, 3, 8, 9, 3 );

**var**

i: Integer;

myTree, wasfound: TTree;

**begin**

myTree := nil;

**for** i := 1 **to** size **do**

**begin**

Insert(myTree, iv[i]);

PrintDown(myTree); WriteLn

**end**;

wasFound := Find(myTree, 7);

**if** wasFound <> **nil** **then** WriteLn('x = ', GetNode(wasFound));

Remove(myTree, 7);

PrintDown(myTree);

**end**.

**Нерекурсивная работа с бинарным деревом**

Для итеративного добавления элемента к бинарному дереву может применяться следующая процедура:

**Procedure** AddIter(Var root: TTree; X: TType);

{ Функция, создающая новый лист дерева с заданным значением Data }

**Function** CreateNode(n: TType): TTree;

**var** p: TTree;

**Begin**

New(p);

p^.Data := n;

p^.Left := nil; p^.Right := nil;

CreateNode := p;

**End**;

**var**

parent, pwalk: TTree;

**Begin**

{ Если корень дерева - нулевой (только начали заполнять дерево, например),

то создаем новый элемент и запоминаем его, как корень }

**if** root = **nil** **then** root := CreateNode(X)

**else** **begin**

{ Если дерево уже не пустое - тогда начинаем "прогулку" по нему... }

pWalk := root; { "гулять" начнем с корня }

**while** pWalk <> **nil** **do**

**begin**

{ пока не добрались до пустого указателя - делаем следующее }

{ запоминаем текущий элемент, в качестве "родителя" его потомка }

parent := pWalk;

{ переходим в левую или правую ветвь в зависимости от соотношения величин

нового элемента и "текущего", которым мы "гуляем" по дереву }

**if** X < pWalk^.data **then** pWalk := pWalk^.left

**else** pWalk := pWalk^.right

**end**;

{ Если мы здесь - значит, добрались до пустого указателя...

Вот теперь делаем то, для чего запоминали родителя текущего элемента:

опять же в зависимости от того, больше или меньше добавляемое значение,

чем значение "родителя", создаем новый элемент и запоминаем его в левой,

или правой ветке... }

**if** X < parent^.data **then** parent^.left := CreateNode(X)

**else** parent^.right := CreateNode(X);

**end**;

**End**;